CS3205 - Introduction to Computer Networks
Even Sem. 2020, Dr. Manikantan Srinivasan
Assignment 3 : HTTP Proxy Implementation
Group Assignment (Max of two members)
Due date: March 9, 2020, 11:59 PM, On Moodle

Extension: 15 % penalty for each 24-hr period; Max. of 48-hrs past the original deadline

February 20, 2020

In this assignment, you will implement a web proxy that passes requests and data between multiple web clients and web servers, concurrently. This is a chance to get to know one of the most popular application protocols on the Internet – the Hypertext Transfer Protocol (HTTP). When you're done with the assignment, you should be able to configure your web browser to use your personal proxy server as a web proxy. Credits Courtesy - https://www.cs.princeton.edu/courses/archive/spring12/cos461/assignments/assignments-proxy.html

1 Introduction: The Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is the protocol used for communication on the web: it defines how your web browser requests resources from a web server and how the server responds. For simplicity, in this assignment, we will be dealing only with version 1.0 of the HTTP protocol, defined in detail in RFC 1945. You may refer to that RFC while completing this assignment, but the instructions should be self-contained.

HTTP communications is in the form of transactions; a transaction consists of a client sending a request to a server and then reading the response. Request and response messages share a common basic format:

- An initial line (a request or response line, as defined below)
- Zero or more header lines
- A blank line (CRLF)
- An optional message body.

The initial line and header lines are each followed by a "carriage-return line-feed" (\r \n) signifying the end-of-line.

For most common HTTP transactions, the protocol boils down to a relatively simple series of steps (important sections of RFC 1945 are in parenthesis):

1. A client creates a connection to the server.

2. The client issues a request by sending a line of text to the server. This request line consists of a HTTP method (most often GET, but POST, PUT, and others are possible), a request URI (like a URL), and the protocol version that the client wants to use (HTTP/1.0). The request line is followed by one or more header lines. The message body of the initial request is typically empty. (5.1-5.2, 8.1-8.3, 10, D.1)

3. The server sends a response message, with its initial line consisting of a status line, indicating if the request was successful. The status line consists of the HTTP version (HTTP/1.0), a response status code (a numerical value that indicates whether or not the request was completed successfully), and a reason phrase, an English-language message providing description of the status code. Just as with the the request message, there can be as many or as few header fields in the response as the server wants to return. Following the CRLF field separator, the message body contains the data requested by the client in the event of a successful request. (6.1-6.2, 9.1-9.5, 10)

4. Once the server has returned the response to the client, it closes the connection.

It's fairly easy to see this process in action without using a web browser. From a Unix prompt, type:

```
telnet www.google.com 80
```

This opens a TCP connection to the server at www.yahoo.com listening on port 80 (the default HTTP port). You should see something like this:

```
Trying 172.217.166.100...
Connected to www.google.com.
Escape character is '\]'.
```

type the following:

```
GET http://www.google.com/ HTTP/1.0 GET http://www.cse.iitm.ac.in/ HTTP/1.0
```

and hit enter twice. You should see something like the following:

```
HTTP/1.0 200 OK
Date:  Thu, 20 Feb 2020 23:15:35 GMT
Expires:  -1
Cache-Control:  private, max-age=0
Content-Type:  text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy!  See g.co/p3phelp for more info."
Server:  gws

(More HTML follows.  Entire capture is given as ref material)
```

There may be some additional pieces of header information as well - setting cookies, instructions to the browser or proxy on caching behavior, etc. What you are seeing is exactly what your web browser sees when it goes to the Google home page: the HTTP status line, the header fields, and finally the HTTP message body- consisting of the HTML that your browser interprets to create a web page. You may notice here that the server responded with HTTP 1.0. Some servers respond with HTTP 1.1 even though you request 1.0 as they refuse to serve HTTP 1.0 content.

2 HTTP Proxies

Ordinarily, HTTP is a client-server protocol. The client (usually your web browser) communicates directly with the server (the web server software). However, in some circumstances it may be useful to introduce an intermediate entity called a proxy. Conceptually, the proxy sits between the client and the server. In the simplest case, instead of sending requests directly to the server the client sends all its requests to the proxy.

The proxy then opens a connection to the server, and passes on the client's request. The proxy receives the reply from the server, and then sends that reply back to the client. Notice that the proxy is essentially acting like both a HTTP client (to the remote server) and a HTTP server (to the initial client).

Why use a proxy? There are a few possible reasons:

- **Performance:** By saving a copy of the pages that it fetches, a proxy can reduce the need to create connections to remote servers. This can reduce the overall delay involved in retrieving a page, particularly if a server is remote or under heavy load.
- **Content Filtering and Transformation:** While in the simplest case the proxy merely fetches a resource without inspecting it, there is nothing that says that a proxy is limited to blindly fetching and serving files. The proxy can inspect the requested URL and selectively block access to certain domains, reformat web pages (for instances, by stripping out images to make a page easier to display on a handheld or other limited-resource client), or perform other transformations and filtering.
- **Privacy:** Normally, web servers log all incoming requests for resources. This information typically includes at least the IP address of the client, the browser or other client program that they are using (called the User-Agent), the date and time, and the requested file. If a client does not wish to have this personally identifiable information recorded, routing HTTP requests through a proxy is one solution. All requests coming from clients using the same proxy appear to come from the IP address and User-Agent of the proxy itself, rather than the individual clients. If a number of clients use the same proxy (say, an entire business or university), it becomes much harder to link a particular HTTP transaction to a single computer or individual.

Links: RFC 1945 The Hypertext Transfer Protocol, version 1.0

3 Assignment Details

3.1 The Basics

Your task is to build a web proxy capable of accepting HTTP requests, forwarding requests to remote (origin) servers, and returning response data to a client. The proxy MUST handle concurrent requests by forking a process for each new client request using the fork() system call. You will only be responsible for implementing the GET method. All other request methods received by the proxy should elicit a "Not Implemented" (501) error (see RFC 1945 section 9.5 - Server Error).

This assignment must be completed in C/C++. It should compile and run (using g++) without errors or warnings, producing a binary called proxy that takes as its first argument a port to listen from. **Do not** use a hard-coded port number.

You shouldn't assume that your server will be running on a particular IP address, or that clients will be coming from a pre-determined IP.

3.2 Listening

When your proxy starts, the first thing that it will need to do is establish a socket connection that it can use to listen for incoming connections. Your proxy should listen on the port specified from the command line and wait for incoming client connections. Each new client request is accepted, and a new process is spawned using fork() to handle the request. There should be a reasonable limit on the number of processes that your proxy can create (e.g., 100). Once a client has connected, the proxy should read data from the client and then check for a properly-formatted HTTP request – but don't worry, you are provided with

libraries that parse the HTTP request lines and headers. Specifically, you will use the libraries to ensure that the proxy receives a request that contains a valid request line:

```
<METHOD> <URL> <HTTP VERSION>
```

All other headers just need to be properly formatted:

```
<HEADER NAME>:  <HEADER VALUE>
```

In this assignment, client requests to the proxy must be in their absolute URI form (see RFC 1945, Section 5.1.2) – as your browser will send if properly configured to explicitly use a proxy (as opposed to a transparent on-path proxies that some ISPs deploy, unbeknownst to their users). An invalid request from the client should be answered with an appropriate error code, i.e. "Bad Request" (400) or "Not Implemented" (501) for valid HTTP methods other than GET. Similarly, if headers are not properly formatted for parsing, your client should also generate a type-400 message.

3.3 Parsing Library

We have provided a parsing library to do string parsing on the header of the request. This library is in proxy_parse.[c—h] in the skeleton code. The library can parse the request into a structure called ParsedRequest which has fields for things like the host name (domain name) and the port. It also parses the custom headers into a set of ParsedHeader structs which each contain a key and value corresponding to the header. You can lookup headers by the key and modify them. The library can also recompile the headers into a string given the information in the structs.

More details as well as sample usage is available in proxy_parse.h, as well as example code on how to use the library. This library can also be used to verify that the headers are in the correct format since the parsing functions return error codes if this is not the case.

3.4 Parsing the URL

Once the proxy sees a valid HTTP request, it will need to parse the requested URL. The proxy needs at least three pieces of information: the requested host and port, and the requested path. See the URL (7) manual page for more info. You will need to parse the absolute URL specified in the request line using the given You can use the parsing library to help you. If the hostname indicated in the absolute URL does not have a port specified, you should use the default HTTP port 80.

3.5 Getting Data from the Remote Server

Once the proxy has parsed the URL, it can make a connection to the requested host (using the appropriate remote port, or the default of 80 if none is specified) and send the HTTP request for the appropriate resource. The proxy should always send the request in the relative URL + Host header format regardless of how the request was received from the client:

Accept from client:

```
GET http://www.cse.iitm.ac.in/ HTTP/1.0
```

Send to remote server:

```
GET / HTTP/1.0
Host:  www.cse.iitm.ac.in
Connection:  close
(Additional client specified headers, if any...)
```

Note: The http message exchange to the cse web site is shared as a .pcap file. The header here generated by browse sends http version 1.1 as default, and the connection is keekp-alive. You can also perform a capture if required for a reference.

Note that **in the assignment** always send HTTP/1.0 flags and a *Connection: close* header to the server, so that it will close the connection after its response is fully transmitted, as opposed to keeping open a persistent connection. So while you should pass the client headers you receive on to the server, you should make sure you replace any *Connection* header received from the client with one specifying *close*, as shown. To add new headers or modify existing ones, use the HTTP Request Parsing Library we provide.

3.6 Returning Data to the Client

After the response from the remote server is received, the proxy should send the response message as-is to the client via the appropriate socket. To be strict, the proxy would be required to ensure a Connection: close is present in the server's response to let the client decide if it should close it's end of the connection after receiving the response. However, checking this is not required in this assignment for the following reasons. First, a well-behaving server would respond with a *Connection: close* anyway given that we ensure that we sent the server a close token. Second, we configure Firefox to always send a *Connection: close* by setting keepalive to false. Finally, we wanted to simplify the assignment so you wouldn't have to parse the server response.

3.7 Testing Your Proxy

Run your client with the following command:

`./proxy <port>`, where `port` is the port number that the proxy should listen on.

As a basic test of functionality, try requesting a page using telnet:

```
telnet localhost <port>
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
GET http://www.google.com/ HTTP/1.0
```

If your proxy is working correctly, the headers and HTML of the Google homepage should be displayed on your terminal screen. Notice here that we request the absolute URL (http://www.google.com/) instead of just the relative URL (/). A good sanity check of proxy behavior would be to compare the HTTP response (headers and body) obtained via your proxy with the response from a direct telnet connection to the remote server. Additionally, try requesting a page using telnet concurrently from two different shells.

For a slightly more complex test, you can configure your web browser to use your proxy server as its web proxy. See the section below for details.

4 Configuring a Web Browser to Use a Proxy

Firefox

Version 10.x:

1. Select Tools–Options (or Edit–Preferences) from the menu.

2. Click on the 'Advanced' icon in the Options dialog.
3. Select the 'Network' tab, and click on 'Settings' in the 'Connections' area.
4. Select 'Manual Proxy Configuration' from the options available. In the boxes, enter the hostname and port where proxy program is running.

Earlier Versions:

1. Upgrade your browser. You're vulnerable to security threats.

To stop using the proxy server, select 'No Proxy' in the connection settings dialog.

Configuring Firefox to use HTTP/1.0

Because Firefox defaults to using HTTP/1.1 and your proxy speaks HTTP/1.0, there are a couple of minor changes that need to be made to Firefox's configuration. Fortunately, Firefox is smart enough to know when it is connecting through a proxy, and has a few special configuration keys that can be used to tweak the browser's behavior.

1. Type 'about:config' in the title bar.
2. In the search/filter bar, type 'network.http.proxy'
3. You should see three keys: network.http.proxy.keepalive, network.http.proxy.pipelining, and network.http.proxy.version.
4. Set keepalive to false. Set version to 1.0. Make sure that pipelining is set to false.

5 Socket Programming

You are already familiar with socket programming. The following information is given for completeness (and also existed as part of the referred material). The Berkeley sockets library is the standard method of creating network systems on Unix. There are a number of functions that you will need to use for this assignment:

- Parsing addresses:
  - inet_ addr : Convert a dotted quad IP address (such as 36.56.0.150) into a 32-bit address.
  - gethostbyname : Convert a hostname (such as argus.stanford.edu) into a 32-bit address.
  - getservbyname : Find the port number associated with a particular service, such as FTP.
- Setting up a connection:
  - socket : Get a descriptor to a socket of the given type
  - connect : Connect to a peer on a given socket
  - getsockname : Get the local address of a socket
- Creating a server socket :
  - bind : Assign an address to a socket
  - listen : Tell a socket to listen for incoming connections
  - accept : Accept an incoming connection

- Communicating over the connection: :
  - read/write : Read and write data to a socket descriptor
  - htons, htonl / ntohs , ntohl : Convert between host and network byte orders (and vice versa) for 16 and 32-bit values

You can find the details of these functions in the Unix man pages (most of them are in section 2) and in the Stevens Unix Network Programming book, particularly chapters 3 and 4. Other sections you may want to browse include the client-server example system in chapter 5 (you will need to write both client and server code for this assignment) and the name and address conversion functions in chapter 9.

6 Multi-Process Programming

In addition to the Berkeley sockets library, there are some functions you will need to use for creating and managing multiple processes: fork, waitpid.

You can find the details of these functions in the Unix man pages:

- *man 2 fork*
- *man 2 waitpid*

Links:

- Guide to Network Programming Using Sockets
- HTTP Made Really Easy- A Practical Guide to Writing Clients and Servers
- Wikipedia page on fork()

A Note on Network Programming

Writing code that will interact with other programs on the Internet is a little different than just writing something for your own use. The general guideline often given for network programs is: be lenient about what you accept, but strict about what you send, also known as Postel's Law. That is, even if a client doesn't do exactly the right thing, you should make a best effort to process their request if it is possible to easily figure out their intent. On the other hand, you should ensure that anything that you send out conforms to the published protocols as closely as possible.

7 What to Submit

The platform for this project will be Linux and C/C++. The libraries provided is to enable you to work with C. If you choose to use C++, you may have to tweak the libraries suitably. Create a tar-gz file with name: Assignment3-RollNo1-RollNo2.tgz (e.g. Assignment3-CS17B099-CS17B098.tgz) that will contain a directory named Assignment3-RollNo1-RollNo2 with all relevant files.

The directory should contain the following files:

- Source Files
- A Makefile which generates all your executables

- A technical REPORT (in PDF format) with help of screen shots showing the working of your implementation. Report your observations and analyze the results, in 1-2 paragraphs. The report should include your name, roll number, assignment number and title.
- The experiments are to be conducted for: a) one client (single terminal) b) simultaneous more than one (3 max) clients.

  Also, give a brief summary as to what you learnt in this experiment and how much beneficial you feel the experiment was.
- a README file containing instructions to compile, run and test your program. README file should be written such that a TA must be able to run your code without your presence/help.

8 Help

1. Ask questions EARLY and start your work NOW. Take advantage of the help of the TAs and the instructor.
2. Submissions PAST the extended deadline SHOULD NOT be mailed to the TAs. Only submissions approved by the instructor or uploaded to Moodle within the deadline will be graded.
3. Demonstration of code execution to the TAs MUST be done using the student's code uploaded on Moodle.
4. NO sharing of code between students, submission of downloaded code (from the Internet, Campus LAN, or anywhere else) is allowed. Code copying will result in a 'U' Course Grade. Students may also be reported to the Campus Disciplinary Committee, which can impose additional penalties.
5. Please protect your Moodle account password. Do not share it with ANYONE. Do not share your academic disk drive space on the Campus LAN.
6. Implement the solutions, step by step. Trying to write the program in one setting may lead to frustration and errors.

9 Grading

- HTTP Proxy working correctly: 70 points (For handling Valid requests 50 points, For other invalid requests 20 points).
- Report: 20 points
- Viva voce: 10 points
- NO README, NO MAKE file: -10 points each